Algorithm

Architecture

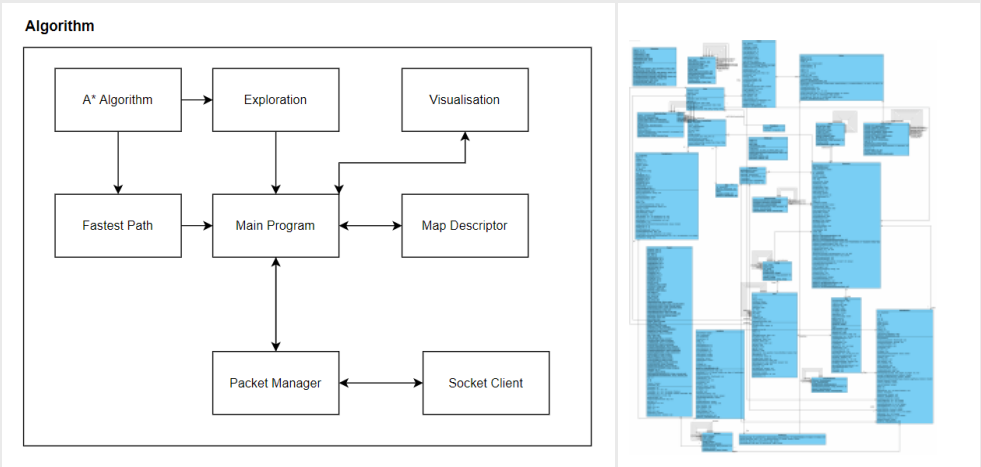


Fig. Architecture Overview (RIGHT), Class Diagram (LEFT) [refer to this link for a clearer image]

Connection with Raspberry Pi

The algorithm uses sockets to establish a connection with the RPi, and then forwards data packets to it over Wi-Fi, which the RPi further processes and forwards to other sub-systems. The algorithm connects to the RPi with IP **192.168.9.9** and port **8081** by using the RPi's dedicated Wi-Fi network, to prevent access from other PCs. The code snippet for establishing the connection is:

```
    public boolean connectToDevice() {
        try {
            InetSocketAddress ISA = new InetSocketAddress(IP_Addr,
Port);
            socket = new Socket();
            socket.connect(ISA, timeout);
        } catch (UnknownHostException u) {
            System.out.println(u);
        }
        return false;
    }
```

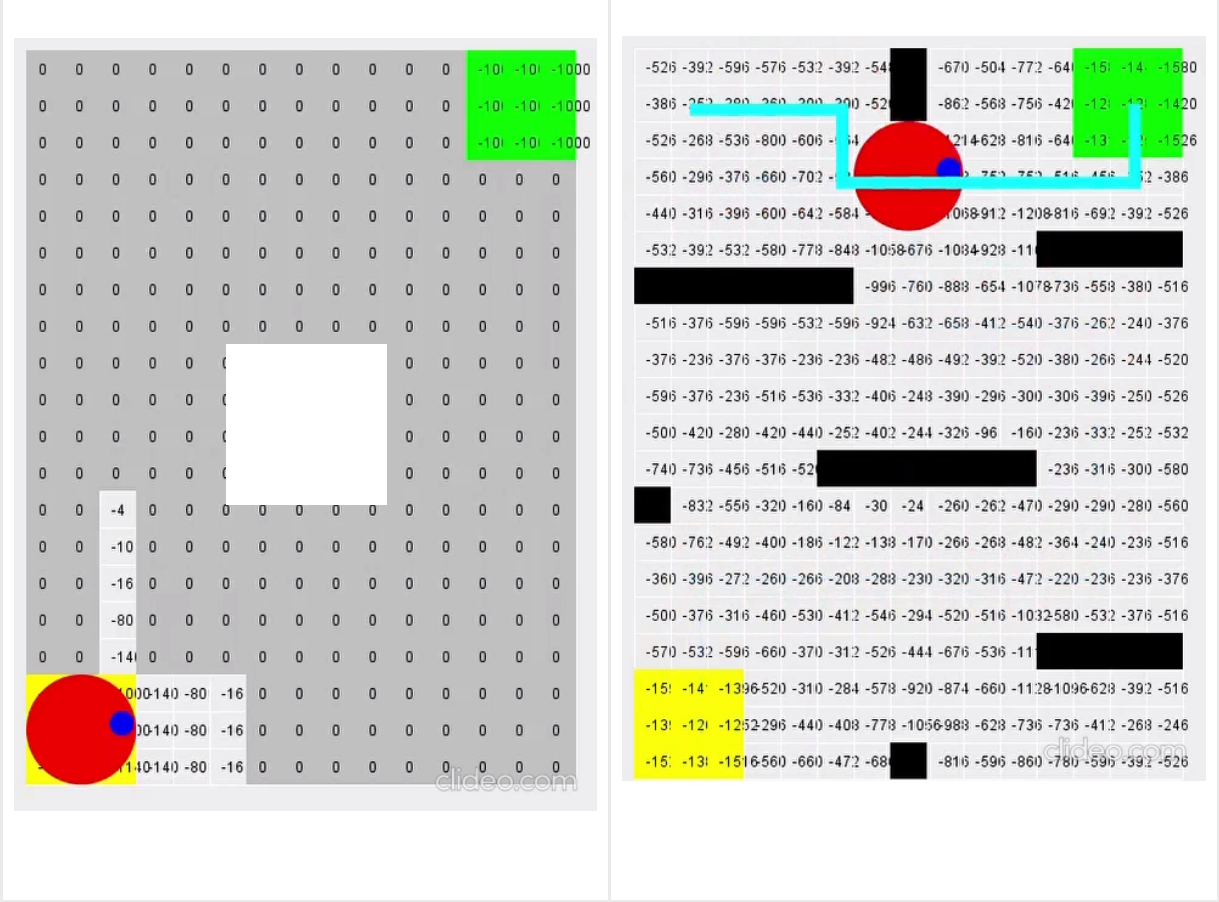


Fig. Exploration Algorithm (RIGHT), Fastest Path (LEFT)

Exploration Algorithm

Our exploration uses the **right wall hugging** algorithm, where every decision that the robot makes, navigates around obstacles and keeps it sticking to the right wall. The decision made by the robot, when it is facing right is as follows (similarly for other directions robot faces in):

- If can move downwards and previously facing right, then **Turn Left**
- If cannot move downwards and can move right, then **Move Forward**
- If cannot move downwards or right, then **Turn Left**
- Else if no wall beside robot, **Back track**

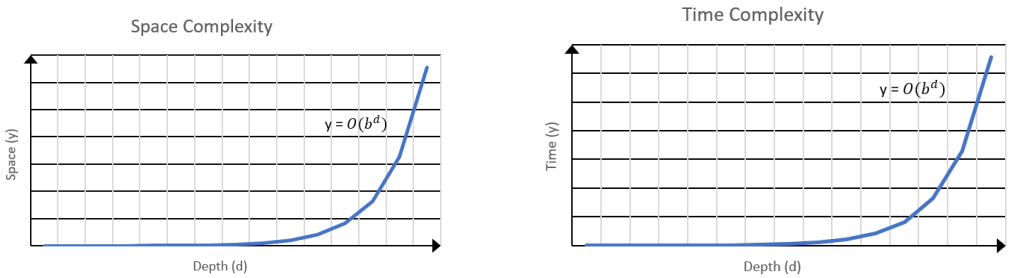
The algorithm uses an enum of 4 main values: EMTPY, OBSTACLE, UNEXPLORED_EMPTY and UNEXPLORED_OBSTACLE. Initially, the entire map is set to unexplored. Once the robot completes hugging the right wall and returns to the start point, it computes the fastest path to the unexplored areas (using **A* search**), explores the type of the unexplored cells, and then makes its way back.

Our algorithm also solves the issue of **phantom blocks** (blocks seen by the sensors but do not actually exist) by overwriting grids, and in the worst case,

backtracking the robot to the last correct location and again performing the same steps.

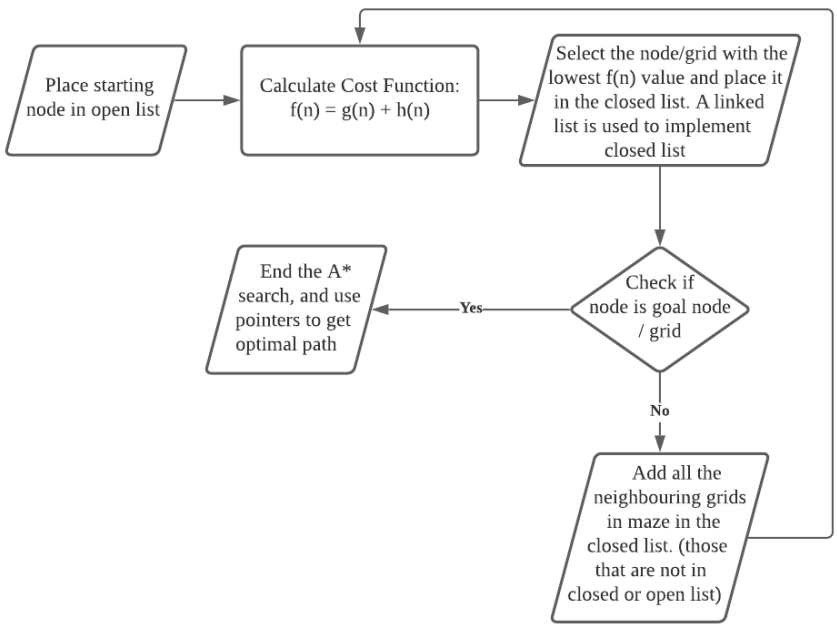
Fastest Path Algorithm

The fastest path algorithm uses the popular **A* search**, a **complete, optimal** graph traversal algorithm, given a start node and a goal node. The space complexity of the A* algorithm is **O(|V|) or O(b^d)**, the number of cells in the arena. The time complexity of the A* search algorithm is **O(|E|) or O(b^d)**, the total number of possible movements to the new cell.



Formulation of cost

Our A* search algorithm calculates the evaluation function *f(n) = g(n) + h(n)* where *g(n)* is the actual path cost to the node *n* and *h(n)* is the heuristic cost from node *n* to the goal node. We have defined *h(n) = (end_x-x) + (end_y-y)*, that is, the sum of the differences in *x* and *y* coordinates from node *n* to the goal node. The A* search chooses the neighboring node with the lowest value evaluation function. Th detailed flowchart for our algorithm is as follows:



To ensure the robot passes through the waypoint, the algorithm first calculates the shortest path from start to the waypoint, and then from the waypoint to the goal node. The following is the main section of the fastest path code:

```
// Iterate through list of node neighbours
for (int i = 0; i < neighbors.size(); i++) {

    // Extract neighbour node information
    Node neighborNode = (Node) neighbors.get(i);
    boolean isOpen = openList.contains(neighborNode);
    boolean isClosed = closedList.contains(neighborNode);
    boolean isObstacle = (neighborNode).isObstacle();
    int clearance = neighborNode.getClearance();
    float costFromStart = node.getCost(neighborNode, goalNode, isStartNode) + 1;

    // Check 1. if node neighbours have not been explored OR 2. if shorter path to
    // neighbour node exists
    if ((!isOpen && !isClosed) || costFromStart < neighborNode.costFromStart) {
        neighborNode.pathParent = node;
        neighborNode.costFromStart = costFromStart;
        neighborNode.estimatedCostToGoal = neighborNode.getEstimatedCost(goalNode);

        // Add neighbour node to openList if 1. node not in openList/closedList AND 2.
        // robot can reach
        if (!isOpen && !isObstacle && size == clearance) {
            openList.add(neighborNode);
        }
    }
}
```